CS3025 Compiladores

Semana 11:
Analisis Semantico (revisited):
Verificacion de Tipos / Typechecking
23 Octubre 2023

Igor Siveroni

El lenguaje IMP con declaraciones de funciones: sintaxis

Hemos estado trabajado con el lenguaje IMP, un lenguaje con declaraciones de funciones globales, definido por la siguiente sintaxis:

```
Program ::= VarDecList FunDecList
VarDecList ::= (VarDec)*
FunDecList ::= (FunDec)+
FunDec ::= ”fun” RType id “(“[ParamDecList]“)” Body “endfun”
ParamDecList ::= Type id (“,” Type id)*
VarDec ::= “var” Type VarList “;”
Type ::= bool | int
Rtype :: bool | int | void
Body ::= VarDecList StmList
StmList ::= Stm (“;” Stm)*
```

Donde un programa es una lista de declaraciones de variables globales seguidas de declaraciones de funciones. La declaración de una función especifica el nombre de la función, tipo de retorno, el nombre y tipos de los parámetros, y el cuerpo definido como un bloque (Body) de declaraciones y sentencias.

El lenguaje IMP: sentencias y expressiones

La gramática de sentencias es:

```
Stm ::=  AssignStm | PrintStm | IfStm | WhileStm | ReturnStm
AssignStm ::= id “=” Exp
PrintStm ::= “print” “(“ Exp “)”
IfStm ::= “if” CExp “then” Body [“else” Body] “endif”
WhileStm ::= “while” Exp “do” Body “endwhile”
ReturnStm ::=“return” “(“ [Exp] “)”
```

La gramática de expresiones, con cuestiones de asociatividad y orden de precedencia resueltas, es:

```
Exp ::=  NumberExp | BoolExp | IdExp | CondExp | FCallExp | ParenthExp | BinExp
NumberExp ::= num     BoolExp ::= “true” | “false”     IdExp ::= id
BinExp ::= Exp op Exp, op in {plus, minus, mul, div, lt, leq, eq }
CondExp ::= “ifexp” “(“ Exp “,” Exp “,” Exp “)”
ParenthExp ::= “(“ Exp “)”
FCallExp ::= id “(“ [ArgList] “)”
ArgList ::= Exp (“,” Exp)*
```

El Sistema de Tipos (type system) de IMP

El sistema de tipos de un lenguaje está definido por lo menos por:

- El conjunto de tipos validos en el lenguaje (tipos base y reglas para generar nuevos tipos)
- Las reglas que determinan si un programa tiene los tipos correctos (equivalencia de tipos, subtyping, inferencia de tipos) i.e. “if a program is correctly typed”.

IMP trabaja con tipos base y tipos de función:

- Los tipos base son `int` y `bool`.
- Los tipos de función están hechos por una lista de tipos base (parámetros) y el tipo de retorno base (que además puede ser `void`).

El Sistema de tipos (type system) de IMP

Denotaremos a los tipos con la variable `T`.

```
T in Type ::= BaseType | FunType
              BaseType ::= int | bool
              FunType ::= BaseType^n x Rtype, n >= 0
Tr in RType :: int | bool | void
```

Escribiremos $T = (T_1,\ldots,T_n) \rightarrow T_r$ para denotar un tipo de función. De este modo podemos escribir, por ejemplo:

- `(int,bool)->int` para denotar el tipo de una función que tom2 2 argumentos, uno de tipo int y otro de tipo bool, y retorna un valor de tipo int.
- `(int,int)->void`: Toma 2 enteros de argumentos y no retorna nada.
- `()->int`: Toma cero argumentos y retorna un entero.
- `()->void`: Cero argumentos y nada de retorno
- Etc.

IMP: Tabla de simbolos / Type Environment

Durante el proceso de verificación de tipos mantendremos siempre a la mano una tabla de símbolos o environment de tipos.

Un environment de tipos mapea nombres de variables a tipos:

`env in TEnv ::= id -> Type` (la -> se usa para definir funciones)

Tendremos en cuenta lo siguiente:

- {} es el environment vacío.
- `env(x)` : retorna el valor de x guardado en env (**lookup**).
- `env(x)` **es valido** si `x in Dom(env)`, es decir, si x esta en el dominio de env.
- `env[x -> T]` actualiza el environment con el nuevo mapping [x -> T] (add_var).
  Entonces: `env[x -> T](x) = T`
  Puede usarse: $\texttt{env[x}_1\texttt{ -> T}_1\texttt{]…[x}_n\texttt{ -> T}_n\texttt{]}$

Asi: `{a->int}[x -> int][a->bool] = {a->bool, x->int }`

IMP: Type checking expressions

Para facilitar la especificación de las reglas de typechecking, expresaremos la sintaxis de IMP usando un tipo especial de sintaxis abstracta (abstract syntax). Empezamos con las expresiones:

```
e in Exp ::=
      | NumberExp(n), n in Int
      | BoolExp(b)  , b in { T, F }
      | IdExp(id)   , id in ID
      | CondExp(e0, e1, e2) |
      | BinExp(e1, e2, op) ,
                op in {plus, minus, mul, div, lt, leq, eq }
      | FCallExp(fname, args) , args in Expn, fname in ID
      | ParenthExp(e)
```

Cuando trabajemos con listas o sequencias, e.g. `args`, podemos usar `|args|= n` para extraer la longitud `n` de la secuencia, o escribir `args = e1,…,en`. Tambien podremos expresar lo siguiente: `forall e in args, P(e).`

IMP: Type checking expresiones

Para especificar el type checking de expresiones definimos la función `tcheck`

```
tcheck: Tenv x Exp -> Type
```

y escribimos `tcheck(env, e) = T` para afirmar que:

- la expresión `e` esta tipeada correctamente (cumple las reglas de tipeo, *correctly typed*) bajo el environment `env`,
- y que, además, `e` tiene el tipo `T`, bajo el environment `env`.

Ahora podemos definir typecheck por casos basados en la sintaxis abstracta de Exp:

```
tcheck(env, NumberExp(n)) = int
```

La implementacion correspondiente en ImpTypeChecker es:

```
ImpType ImpTypeChecker::visit(NumberExp* e) {
    return inttype;
}
```

IMP: Type checking expresiones

Procedemos del mismo modo con las constantes booleanas:

```
tcheck(env, BoolExp(b)) = bool
```

Con código

```
ImpType ImpTypeChecker::visit(TrueFalseExp* e) {
  return booltype;
}
```

Y los identificadores,

```
tcheck(env, IdExp(id)) = env(x)
```

Con codigo

```
ImpType ImpTypeChecker::visit(IdExp* e) {
  if (env.check(e->id)) // esta e->id en env??
    return env.lookup(e->id);
  else {
    cout << "Variable indefinida: " << e->id << endl;
    exit(0);
  }
}
```

IMP: Type checking ParenthExp y CondExp

```
tcheck(env, ParenthExp(e)) = tcheck(env, e)
```

Codigo:

```
ImpType ImpTypeChecker::visit(ParenthExp* ep) {
  return ep->e->accept(this);
}
```

```
tcheck(env, CondExp(e0, e1, e2)) = T ifi
tcheck(env,e0) = bool && T = tcheck(env,e1) = tcheck(env,e2)
```

Codigo:

```
ImpType ImpTypeChecker::visit(CondExp* e) {
if (!e->cond->accept(this).match(booltype)) {
  typerror("Tipo en ifexp debe de ser bool”);
}
ImpType ttype =  e->etrue->accept(this);
if (!ttype.match(e->efalse->accept(this))) {
  typerror("Tipos en ifexp deben de ser iguales”);
}
  return ttype;
}
```

IMP: Type checking BinExp

tcheck(BinExp(e1,e2,op)) = int

ifi

tcheck(env,e1) = int &&
tcheck(env, e2) = int
op in {plus,minus,mul,div}

tcheck(BinExp(e1,e2,op)) = bool

ifi

tcheck(env,e1) = int &&
tcheck(env, e2) = int
op in {lt, leq, eq }

```
ImpType ImpTypeChecker::visit(BinaryExp* e) {
  ImpType t1 = e->left->accept(this);
  ImpType t2 = e->right->accept(this);
  if (!t1.match(inttype) || !t2.match(inttype)) {
    cout << "Tipos en BinExp deben de ser int" << endl;
    exit(0);
  }
  ImpType result;
  switch(e->op) {
  case PLUS: case MINUS:
  case MULT: case DIV:
  case EXP:
    result = inttype;
    break;
  case LT: case LTEQ:
  case EQ:
    result = booltype;
    break;
  }
  return result;
}
```

IMP: Type checking statements (sentencias)

Del mismo modo que lo hecho con expresiones, expresaremos la sintaxis de las sentencias usando sintaxis abstracta (abstract syntax):

```
s in Stm ::=
       | AssignStm(id,e)
       | PrintStm(e)
       | IfStm(e, bd1, bd2) , bd1, bd2 in Body
       | WhileStm(e, bd)    , bd in Body
       | ReturnExp(e)

b in Body    ::= Body(vdlist, slist) , vdlist in VardDec*
vd in VarDec ::= (T, id)
```

IMP: Type checking statements (reglas)

Para el caso de sentencias y bloques (Body), definimos el predicado tcheck de la siguiente manera:

```
tcheck: TEnv x Stm -> Bool        tcheck: TEnv x Body -> Bool
```

Y escribimos `tcheck(env, s)` si:

- La sentencia `s` satisface las reglas de tipos, dado el environment `env`.
  Igual para `tcheck(env, bd))`

Especificamos `tcheck` para todas las sentencias por casos basado en la sintaxis de `Stm`:

```
tcheck(env, PrintStm(e)) ifi tcheck(env,e)
```

Y presentamos el código respectivo en ImpTypeChecker:

```
void ImpTypeChecker::visit(PrintStatement* s) {
  s->e->accept(this);
  return;
}
```

IMP: Type checking AssignStm

```
tcheck(env, AssignStm(id,e)) ifi tcheck(env,e) = env(id)
```

Con código

```
void ImpTypeChecker::visit(AssignStatement* s) {
  ImpType type = s->rhs->accept(this);
  if (!env.check(s->id)) {
    cout << "Variable " << s->id << " undefined" << endl;
    exit(0);
  }
  ImpType var_type = env.lookup(s->id);
  if (!type.match(var_type)) {
    cout << "Tipo incorrecto en Assign a " << s->id << endl;
  }
  return;
}
```

IMP: Type checking WhileStm

```
                tcheck(env, IfStm(e,bd1,bd2))
                          ifi
tcheck(env,e0) = bool && tcheck(env,bd1) && tcheck(env,bd2)
```

Con código:

```
void ImpTypeChecker::visit(IfStatement* s) {
  if (!s->cond->accept(this).match(booltype)) {
    typerror("Expresion conditional en IF debe de ser bool”);
  }
  s->tbody->accept(this);
  if (s->fbody != NULL)
    s->fbody->accept(this);
  return;
}
```

```
Assumiendo tcheck(env,null) = true
```

IMP: Type checking WhileStm

```
tcheck(env, WhileStm(e, bd))
                ifi
tcheck(env,e) = bool && tcheck(env,bd)
```

Con código:

```
void ImpTypeChecker::visit(WhileStatement* s) {
  if (!s->cond->accept(this).match(booltype)) {
    typerror("Expresion conditional en IF debe de ser bool”);
  }
  s->body->accept(this);
  return;
  }
```

IMP: Type checking Body

tcheck(env, Body(nil, slist)) **ifi** forall s in slist: typecheck(env, s)

tcheck(env, Body(vdlist, slist))
ifi
vdlist =(T_1,id_1),…,(T_n,id_n), T_i in {int, bool}
env1 = env[id_1 -> T_1],…,[id_n -> T_n]
forall s in slist: typecheck(env1, s)

Con código:

```
void ImpTypeChecker::visit(Body* b) {
  env.add_level();
  b->var_decs->accept(this);
  b->slist->accept(this);
  env.remove_level();
  return;
}
```

IMP: Type checking Body

comparar

$$vdlist = (T_1, id_1), \ldots, (T_n, id_n),\ T_i \text{ in } \{int, bool\}$$
$$env1 = env[id_1 \rightarrow T_1], \ldots, [id_n \rightarrow T_n]$$

Con el código:

```
void ImpTypeChecker::visit(VarDec* vd) {
  ImpType type;
  type.set_basic_type(vd->type);
  if (type.ttype==ImpType::NOTYPE || type.ttype==ImpType::VOID) {
    cout << "Tipo invalido: " << vd->type << endl;
    exit(0);
  }
  list<string>::iterator it;
  for (it = vd->vars.begin(); it != vd->vars.end(); ++it) {
    env.add_var(*it, type);
  }
  return;
}
```

IMP: Type checking Body

comparar

```
forall s in slist: typecheck(env1, s)
```

Con el código:

```
void ImpTypeChecker::visit(StatementList* s) {
  list<Stm*>::iterator it;
  for (it = s->slist.begin(); it != s->slist.end(); ++it) {
    (*it)->accept(this);
  }
  return;
}
```

Los casos para declaraciones de funciones, return y llamadas de funciones siguen la misma lógica.